Description

Method and arrangement for modifying software or source code

The invention relates to a method and an arrangement for modifying software or source code wherein a piece of software or a source code is converted into a representation in a meta markup language, for example XML, transformed there, for example using XSLT, and then said transformed representation formulated in the meta markup language is converted back into a modified piece of software or into a modified source code, for example in the same source language.

Although several possibilities for making subsequent changes or modifications to software are known from the prior art, they all have a number of disadvantages compared with the invention:

One possibility of influencing software is realized with the aid of parameterization. Configuration files are typically used for parameterization and storage of application-specific "parameter data". The structure and format of said files are defined during the development phase and are in no way modifiable once the software has been shipped.

Plug-ins open up the possibility of extending already "shipped", compiled software with functionality characteristics. Toward that end it is necessary that the structures for integrating and using plug-ins are already implemented or specified during the development phase (interfaces,...).

Code generators generate source or binary code with the aid of templates which are completed at predetermined points, for example by means of parameters that have been passed. In this way it becomes possible to generate different software for different customers for example, said software differing at

precisely defined points. In this case, however, only special points (and not arbitrary points) in the code can be modified, which points must be specified precisely when the template is produced. Code generators are typically used on the developer side.

A special application possibility of using variation points in the form of updates/patches is known from US patent application US 6052531A1.

A Java source code transformation tool called BeautyJ is known from the internet at http://beautyj.berlios.de/, wherein a Java source code is converted into an XML representation, "beautified" using Sourclet API, for example by insertion of white spaces or modified comments at specific points, and subsequently the modified source code can be converted back into Java source code. A transformation by means of XSLT is only proposed here, for this purpose, but not implemented.

The object on which the invention is based is therefore to specify a method and an arrangement for modifying source code wherein a more extensive, more flexible and more efficient modification of the software or the source code is achieved.

This object is achieved according to the invention by the features of claims 1, 6, 8 and 13 with regard to the method and by the features of claims 22 to 26 with regard to the arrangement. The remaining claims relate to preferred embodiments of the invention.

The invention is essentially characterized in that, in a first variant, selected components of a piece of software serve as variation points, whereby said variation points are converted into a first code formulated in a meta markup language, for example XLML, the software, now in mixed form, is shipped, and the first code is converted at the customer end by means of one or more transformations, for example XSLT,

exclusively in accordance with transformation rules into a second code formulated in the meta markup language,
that, in a second variant, a first code containing at least one language extension and formulated in a meta markup language is converted in accordance with transformation rules into a more easily verifiable second code formulated in the meta markup language without said language extensions,
that, in a third variant, a source code transformed into a meta markup language is transformed in such a way that, following a conversion back into the original programming language, a new source code is produced in which not only the representation but also the actual program content and/or the functionality has been changed, or
that, in a fourth variant, a source code transformed into a meta markup language with, for example, initial states, code fragments to be replaced and foreign language modules tailored to the respective natural language of the user is mixed by means of transformation, as a result of which, following a back-conversion, a new source code is produced in which not only the representation, but also the actual program content and/or the functionality has been changed.

The invention will be explained in more detail below with reference to the examples illustrated in the drawings, in which:

Figure 1 shows an overall block diagram serving to explain a first variant of the invention,

Figure 2 shows an overall block diagram serving to explain a second variant of the invention,

Figure 3 shows a block diagram serving to explain the inventive transformation of pre-processing extensions,

Figure 4 shows an overall block diagram serving to explain a third variant of the invention,

Figure 5 shows a block diagram serving to explain the inventive modification through the use of aspects,

Figure 6 shows a block diagram serving to explain the inventive insertion of migration functionality,

Figure 7 shows a block diagram serving to explain the inventive modification through the use of templates, filters and patterns,

Figure 8 shows an overall block diagram serving to explain a fourth variant of the invention,

Figure 9 shows a block diagram serving to explain the inventive replacement of code fragments,

Figure 10 shows a block diagram serving to explain the inventive insertion of state data,

Figure 11 shows a block diagram serving to explain the possibilities for variation of the inventive incorporation of information and

Figure 12 shows a block diagram serving to explain the inventive incorporation of foreign language modules for internationalizing the source code.

1st variant of the invention

Figure 1 shows an overall block diagram serving to explain the invention, wherein a piece of software **SW** consisting of source text **SC1**, **SC2** and **SC** is initially converted into a shippable piece of software **SW***, with certain parts of the software such as, for example, **SC1** now being available as binary code/byte code **B1** and other parts such as, for example, **SC2** being

converted by a converter **CONV** into a first code **CodeML** formulated in a meta markup language, such that henceforth they form variation points **VP**, for example **VP1**, in the executable software **SW***. Said software **SW*** can be modified prior to or at runtime in such a way that the code **VP**, for example **VP2**, represented in the meta markup language is converted by means of a transformation **T** and transformation rules **TR** into a second code **CodeML*** formulated in the meta markup language, which code is now either present in **SW*** as a modified variation point, for example **VP2***, or following the transformation **T** is transformed by means of a converter **RCONV** into a source code **SC*** and then converted by means of **COMP** into a byte code/binary code **VP2B***. In both cases **SW** and **SW*** differ at the locations of the variation points and in this way can be tailored to specific requirements (for example toolkit replacement, updates, etc.).

The codes **CodeML** and **CodeML*** or **VP** and **VP*** are formulated for example in the meta markup language XML, where "XML" stands for Extensible Markup Language.

It is of particular advantage here that this does not have to be carried out by the program developer, but can be accomplished independently by the appropriately equipped and knowledgeable customer. For this purpose an operator or administrator on the customer side needs only to apply an appropriate transformation **T** with the requisite substitution, modification and removal rules **TR** in order to adapt the software to their specific requirements or, as the case may be, to carry out an update or patching. During the updating or patching of software adapted to specific customer requirements there have frequently been problems in the past due to inconsistencies which can be avoided from the outset by this invention and the possibility of pipeline application or ordered sequential execution.

The program listings **Listing 1** to **Listing 5** contained in **Annex 1** show this on the basis of an actual example:

Typically, a software shipment to two different customers can use different toolkits which differ from each other in terms of performance, price etc.

Thus, in this case a code which originally uses a registration class
import electric.registry.Registry
from a **Glue toolkit** now uses two new "registration classes" for the second customer
import org.apache.axis.client.Call and
import org.apache.axis.client.Service from an **Axis toolkit**.

In XSL this can happen, for example, by means of

```
<xsl:template match="import">
    <xsl:if test="dot/name='Registry'">
     <import>
       <dot>
         <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
         <name>client</name></dot><name>Call</name>
       </dot>
     </import>
     <import>
       <dot>
         <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
           <name>client</name></dot><name>Service</name>
       </dot>
     </import>
    </xsl:if>
    <xsl:if test="dot/name!='Registry'">
      <xsl:copy-of select="."/>
    </xsl:if>
    ...

  </xsl:template>
```

In XSL, templates are applied to the pattern defined in **match**. The import template in the listing example is therefore applied to all the original import statements. In the actual example it simply ignores all the original GLUE registry imports and instead inserts the two Axis-specific imports.

The method according to the invention further results in a number of additional advantages such as, for example:

1. Only one system is required for issues such as patching, customizing, updating, etc., and not a series of different, in some cases proprietary tools.

2. The method is based on standards such as XML and XSLT and in terms of convertibility into other programming languages is subject to fewer restrictions than other methods for modifying source code.

3. No proprietary special-purpose solutions are required even for special and complicated source code modifications, but instead existing standards such as **XSLT, XPath** and **XQuery** can be used for this purpose.

4. This type of modification permits the setting up of hierarchies, among other things through the possibility for ordered, automated sequential execution (pipelines) of multiple transformations, of patches for example.

5. The transformations can be stored in XSLT files for general reuse, enabling libraries to be produced for example for specific execution sequences.

6. An XML representation of the source code can be stored in an XML database and when necessary easily adapted with the aid of an XSLT to the individual customer requirements (customization).

7. Through the use of the **XMLSchema** or **DTD** standards or appropriate XSLTs the code can be checked in advance (without compilation) for specific correctness aspects (validated).

8. Standard XML tools can be used for simple processing or visualization and determination of relationships in the code.

9. Permanent XML-based program libraries which support XPath queries can improve the reuse of code through more efficient retrieval of a code or code fragments or templates.

2nd variant of the invention

Figure 2 shows an overall block diagram serving to explain the invention wherein a first code **CodeML** which is formulated in a meta markup language, contains a language extension **LE** and cannot be converted into valid source text **SC*** by **RCONV** is converted by a transformation **T** in accordance with transformation rules **TR** which contain a language converter **LC** into a second code **CodeML*** which is formulated in the meta markup language, contains none of said language extensions **LE** and can therefore be converted into a source code **SC*** which can be converted in turn by means of a compiler **COMP** into valid binary code/byte code **B***.

The modified source code **SC*** is formulated for example in the Java programming language and the codes **CodeML** and **CodeML*** are formulated for example in the meta markup language XML, where "XML" stands for Extensible Markup Language.

The transformation **T**, for example an Extended Stylesheet Language Transformation or **XSLT,** is described by means of transformation rules **TR**, for example within **XSL** (Extended Stylesheet Language) files, with, for example, the rules formulated in XSL being used among other things as a language converter LC and describing how the XML-coded source code **CodeML** with a language extension **LE** can be transformed into a variant without language extension.

Figure 3 shows a first exemplary embodiment in which a first code **CodeML** formulated in a meta markup language contains a language extension for pre-processing **PPE** (e.g. <define>, <ifdef>, etc.) and is transformed with the aid of a transformation **T** in accordance with transformation rules **TR** which possess a pre-processing language converter **PPLC** which resolves or uses the **PPE** into a second code CodeML* formulated in the meta markup language and without language extension.

The language extension is typically embodied in the form of elements for generic programming **1** and/or for pre-processing **2** and/or a customer- or developer-specific grammar **3** and/or for macros **4**.

All language extensions or the code CodeML itself can be validated at any time through the use of document type definitions (DTDs) or XMLSchema.

The invention provides the programmer with greater freedom, since the grammar of the programming language used can be adapted to the programmer's requirements and a retransformation back to the normal grammar of the programming language only has to be performed at the end of the program development phase. A significant advantage also consists in the fact that a validation of the language extensions can be performed using a compiler provided for the normal programming language.

The program listings **Listing 1** to **Listing 3** contained in **Annex 2** show the resolution of the pre-processing extensions **PPE** on the basis of an actual example in which the class **TestOutput.xjava** represented in **Listing 1** contains a **PPE** in the form of <define name="m" value="private">, which has an effect on the values of the <m> elements, and is now converted by means of a transformation **T** in accordance with transformation rules **TR** (in this case: **PPLC**) into the XML-based form **TestOutput.xjava*** represented in **Listing 2**, in which all <m> elements are replaced by a <private/> element determined via value="private". By this means it is possible to convert **TestOutput.xjava*** into the source code **TestOutput.java** shown in **Listing 3**.

The method according to the invention further results in a number of additional advantages such as, for example:

1. Only one system is required for issues such as customizing of programming languages, etc., and not a series of different, in some cases proprietary tools.

2. The method is based on standards such as XML and XSLT and in terms of convertibility into other programming languages is subject to fewer restrictions than other methods for modifying source code.

3. No proprietary special-purpose solutions are required even for special and complicated source code modifications, but instead existing standards such as **XSLT, XPath** and **XQuery** can be used for this purpose.

4. This type of modification permits the setting up of hierarchies, among other things through the possibility for ordered, automated sequential execution (pipelines) of multiple transformations, of language adaptations for example.

5. The transformations can be stored in XSLT files for general reuse, enabling libraries to be produced for example for specific execution sequences.

6. An XML representation of the source code can be stored in an XML database and when necessary easily adapted with the aid of an XSLT to the individual customer or, as the case may be, developer requirements (customization).

7. Through the use of the **XMLSchema** or **DTD** standards or appropriate XSLTs the code can be checked in advance (without compilation) for specific correctness aspects (validated).

8. Standard XML tools can be used for simple processing or visualization and determination of relationships in the code.

<u>3rd variant of the invention</u>

Figure 4 shows an overall block diagram serving to explain the invention wherein a source code **SC** is initially converted by a converter **CONV** into a first code **CodeML** formulated in a meta markup language, whereby the source code **SC**, if compiled immediately, would yield a byte code or binary code **B**. The code **CodeML** represented in the meta markup language is now modified by means of a transformation **T** exclusively through the use of transformation rules **TR** which consist of conditions **C** and/or logic **L** and/or code fragments **CF**, thereby resulting in a second code **CodeML*** also formulated in the meta markup language. Following the transformation, a further converter **RCONV** converts the code **CodeML*** back into a source code **SC*** which typically is formulated in the same language as the source code **SC**. Finally, the modified code **SC*** is converted by a compiler **COMP** into a modified byte code **B*** or else immediately into an executable binary code. A significant aspect here is that the byte code **B*** is different in principle from the byte code **B** or that the source code has been changed not only in terms of its representation, but also in terms of its program execution.

The source code **SC** and the modified source code **SC*** are formulated for example in the Java programming language and the codes **CodeML** and **CodeML*** are formulated for example in the meta markup language XML, where "XML" stands for Extensible Markup Language.

The transformation **T**, for example an Extended Stylesheet Language Transformation or **XSLT,** is described by means of transformation rules **TR**, for example within **XSL** (Extended Stylesheet Language) files, with, for example, the rules **TR** formulated in XSL describing among other things how the source code **CodeML** coded in XML is combined with the code fragment **CF** in order to form a new modified source code **CodeML*** with integrated **CF,** or a variation thereof which can now contain, for example, additional logging functionality.

Figure 5 shows a first exemplary embodiment in which the transformation rules **TR** correspond specifically to aspect rules **AR** conforming to aspect-oriented programming (AOP) which, expressed in the AspectJ language, contain at least one point cut **PC** and/or at least one advice type **AT** and/or at least one advice body **AB** and in their sequence can be assigned to the components from **Figure 5**.

In this way a (tool-independent) AOP can be implemented which, compared to other solution variants, for example AspectJ, generates no additional overhead in the generated code **CodeML*** and is not subject to the usual restrictions (extra compiler, syntax, etc.) of existing aspect languages.

An aspect is the term applied in AOP to an entity which modularizes crosscutting concerns, e.g. logging, and encapsulates them at one location. The corresponding code, which previously ran through a plurality of modules, is in this case merged with the aid of a single aspect.

The program listings **Listing 1** to **Listing 5** contained in **Annex 3** shows this on the basis of an actual example in which the file TestCalculator.java contained in Listing 1 is initially converted into an XML representation TestCalculator.xjava. **Listing 3** contains the description of an aspect in the form of a file LoggingAspect.xsl which contains all the necessary transformation rules and ensures that each method bearing a "cal" in its name is found and a print command **System.out.println("calculate begin")** is inserted at the beginning of the execution of said method and a print command **System.out.println("calculate end")** is inserted at the end of the execution of said method.

If, for example, all methods matching the pattern **"cal"**, in other words e.g. public String calcValues() or similar, are to be caused to actuate a system output upon entry and exit in all 151 classes of a project, then first

```
match="*[(name()='curly')and(ancestor::method[contains(name,'cal')])]"
```

is used to select all methods with the "cal" pattern, next

```
<expr>
 <paren>
  <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
  <exprs>
   <expr>
    <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> begin"</xsl:text>
   </expr>
  </exprs>
 </paren>
</expr>
```

is used to insert a statement "**System.out.println(%Name of the Method% + " begin")**", e.g. **System.out.println("calculate end")**, then

```
<xsl:copy-of select="*" />
```

is used to insert the original code of the method, and finally

```
<expr>
 <paren>
  <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
  <exprs>
   <expr>
    <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> end"</xsl:text>
   </expr>
  </exprs>
 </paren>
</expr>
```

is used to insert a statement "**System.out.println(%Name of the Method% + " end")**", e.g. **System.out.println("calculate end")**.

Thus, instead of initiating a corresponding logging output in all 151 classes, this can take place in this case within one logging aspect at one location. Accordingly, modifications also only have to be made at one location.

Figure 6 relates to a second exemplary application of the invention wherein a transformed code **CodeML*** is likewise generated from a source code **CodeML** by means of the transformation **T**, which transformed code **CodeML*** now contains a mechanism for backing up (OLD) or, as the case may be, determining (NEW) at least one state for the desired (version)

migration. In this case the transformation rules **TR** are embodied in such a way that they can be designated as migration rules **MR** and as well as **C** and **L** additionally contain at least one fragment, referred to as checkpoints **CP**, for generating (**CP Write**) or reading in (**CP Read**) states (**CP Data**) which enable a migration from an older version **B*OLD** to a newer version **B*NEW**.
The format conversions of the system states to be transferred, which format conversions are required for a migration, can also be taken into account by this means. Because of this, future migrations do not have to be taken into account as early as in the preliminary phase, as a result of which the test overhead and related potential program bugs are avoided in early program versions.
By automating the migration human errors are avoided since the migration is performed much more systematically.

Figure 7 shows a third exemplary embodiment in several sub-variants wherein a source code **CodeML** coded in XML is likewise converted by means of a transformation **T** into a modified **CodeML***. In this case, however, the transformation **T** is effected by means of transformation rules **TR** which in each variant consist of at least **C** and **L** and as in the case of the conversion of templates **TP** additionally include at least one template fragment **TPF**, for example for the conversion into an **EJB** (Enterprise Java Bean) and in the case of the conversion of patterns **P** possess at least one pattern fragment **PF**, for example for the use of proxy, factory or singleton patterns. For the implementation of filters **FI**, **C** and **L** are sufficient, since here only code is removed and so, for example, superfluous output statements or comments can be eliminated.

Through the appropriate use of **proxy patterns** local calls can be converted into remote calls or in a similar manner local classes can be converted into EJB (Enterprise Java Beans) classes.

Conversely, a valid template **TP** which can be used as a template for other source code can also be generated from the XML-coded source code **JavaML** or a fragment of this code with the aid of a transformation **T** and appropriate rules **TR**.

The aforementioned embodiments of the method according to the invention can be implemented individually and in any order sequentially.

The method according to the invention further results in a number of additional advantages such as, for example:

1. Modifications to the source code can be made quickly and flexibly.

2. Only one system is required for issues such as pattern application, migration, AOP, filtering, etc., and not a series of different, in some cases proprietary tools.

3. The method is based on standards such as XML and XSLT and in terms of convertibility into other programming languages is subject to fewer restrictions than other methods for modifying source code.

4. No proprietary special-purpose solutions are required even for special and complicated source code modifications, but instead existing standards such as **XSLT, XPath** and **XQuery** can be used for this purpose.

5. This type of modification permits the setting up of hierarchies, among other things through the possibility for sequential execution (pipelines) of multiple transformations.

6. The transformations can be stored in XSLT files as general transformations for reuse, enabling libraries to be produced for example for specific execution sequences.

7. An XML representation of the source code can be stored in an XML database and when necessary easily adapted with the aid of an XSLT to the individual customer requirements (customization).

8. Through the use of the **XMLSchema** or **DTD** standards or appropriate XSLTs the code can be checked in advance (without compilation) for specific correctness aspects (validated).

9. Standard XML visualization tools can be used for simple processing or visualization and determination of relationships in the code.

10. Permanent XML-based program libraries which support XPath queries can improve the reuse of code through more efficient retrieval of a code or code fragments or templates.

4th variant of the invention

Figure 8 shows an overall block diagram serving to explain the invention wherein a source code **SC** is initially converted by a converter **CONV** into a first code **CodeML** formulated in a meta markup language, whereby the source code **SC**, if compiled immediately, can produce a byte code or binary code **B**. As well as the code **CodeML** represented in the meta markup language, an additional item of information **INFO** is now added by means of a transformation **T** described by means of transformation rules **TR** to the code **CodeML** or finally to the source code **SC**, thereby yielding a second code **CodeML*** likewise formulated in the meta markup language. Following the transformation, a further converter **RCONV** converts the code **CodeML*** back into a source code **SC*** which typically is formulated in the same language as the source code **SC**. Finally, the modified code **SC*** is converted by means of a compiler **COMP** into a modified byte

code **B*** or else immediately into an executable binary code. A significant aspect here is that the byte code **B*** is different from the byte code **B** or that the source code has been changed not only in terms of its representation, but also in terms of its program execution.

The source code **SC** and the modified source code **SC*** are formulated for example in the Java programming language and the codes **CodeML** and **CodeML*** are formulated for example in the meta markup language XML, where "XML" stands for Extensible Markup Language.

Figure 10 shows a first exemplary embodiment in which the additional information **INFO** is additionally merged with the **CodeML** in the form of data **D**, for example initialization states **SSDb**, state data **SDa**, database data **Dc**, arbitrary data **x**, said data representing for example fixed states or values for constants and variables. In this way the source code **SC** can be supplied with fixed states and transformed such that a required state is available immediately at program runtime, e.g. as initialization state **SSDb**, and no longer has to be determined separately. In this way object states can also be incorporated into the code, said object states enabling a recovery of an interrupted program at the same location with the same states, without the necessity for additional complicated and time-consuming programming measures to be taken for this purpose.

The transformation **T**, for example an Extended Stylesheet Language Transformation or **XSLT**, is described by means of transformation rules **TR**, for example within **XSL** (Extended Stylesheet Language) files, with the rules formulated in XSL, for example, describing among other things how the source code **CodeML** coded in XML is combined with the state data from **D** in order to form a new modified source code **CodeML*** with **SSDb**, **SDa** and **Dc**.

The rules of a transformation **T** can be embodied in such a way here that the information is additionally merged in its original form but also in a form modified by means of rules.

The rules of a transformation **T** can also be embodied in such a way here that the transformation **T** is influenced by the information or data, for example with the aid of if conditions.

The program listings **Listing 1** to **Listing 6** contained in **Annex 4** show this on the basis of an actual example in which the uninitialized variable **String m_sWelcome** is transformed into an initialized form **String m_sWelcome = "hello";** in a test class of the source code. In this case **Listing 1** shows the corresponding Java program TestOutput.java, which is converted into an XML representation TestOutput.xjava. The XML file State.XML is represented with the state value **"hello"** in **Listing 3**. In **Listing 4** there then follows the transformation statement for mixing **Mixing.xsl** which ensures by means of statements such as **template match =** and **apply-templates** that the code is modified at the right location.

Figure 9 shows a second exemplary embodiment in which a code fragment **CFb** coded in XML with an original source code **CodeML** coded in XML which contains a code fragment **CFa** is transformed by means of the transformation **T** in such a way that a code fragment **CFb** is contained in the modified XML-coded source code **CodeML*** in place of the previously present fragment **CFa**. In this case the transformation **T** is also controlled by transformation rules **TR**. A replacement of code fragments of said kind can be referred to in certain situations, for example, as "patching". The method according to the invention enables patching to be accomplished in a consistent manner with a maximum degree of freedom for the software developer, it being possible for example to implement said patching automatically and taking mutual dependencies into account.

An actual case for this exemplary embodiment is illustrated by means of the **Listings 1A to 6A** of the program listings contained in **Annex 5**. A **TextOutput.xjava** is generated in turn from the Java source code **TestOutput.java**.
Listing 3A contains the file **CodeFragment.xml**, which provides a code fragment. **Listing 4A** now contains the rules for the transformation **T** in the file **Patching.xsl**, with the commands **template match =** and **apply-templates** being used in turn. **Listing 5A** then lists the contents of the file **TestOutput.xjava(*)** with the modified XML source code and the modified Java source code is represented in the file **TestOutput.java(*)** in **Listing 6A**. In this example the string assignment **String m_sWelcome = "hello";** in the public test class is replaced by a string assignment **String m_sWelcome = System.getProperty("WELCOME");**, whereby in this case therefore the fixed value "hello" is replaced by the attribute "WELCOME" required by the system and the, for example, erroneously "hard-coded" value assignment can now be "patched".

Figure 11 relates to a third exemplary application of the invention wherein the information **INFO** from **drawing 1** is not only merged in addition in the above-specified manner in the form of information **INFO1**, but also in addition in the form of information **INFO2** or fragments embedded in the transformation rules.

Figure 12 relates to a fourth exemplary application of the invention wherein XML source code **CodeML** is combined with the code fragment **CF**, which contains the foreign language fragments **LF1** and **LF2,** by means of the transformation **T** in order to obtain a modified code **CodeML*,** for example tailored to the natural language of the user (L1=german). In this case the transformation **XSLT** is determined by transformation rules **TR** which specify the points in the source code that are to be changed as well as the respective chosen natural language, in other words, for example, **german** or **english**. Thus, by means of the method according to the invention a localization and internationalization of the source code is achieved in an

efficient and economical manner, with the additional runtime required for this purpose being reduced to a minimum.

The aforementioned embodiments of the method according to the invention can be implemented individually and in any order sequentially.

The method according to the invention results in a number of advantages such as, for example:

1. Modifications to the source code can be made quickly and flexibly.

2. Only one system is required for issues such as patching, customizing, updating, etc., and not a series of different, in some cases proprietary tools.

3. The method is based on standards such as XML and XSLT and in terms of convertibility into other programming languages is subject to fewer restrictions than other methods for modifying source code.

4. No proprietary special-purpose solutions are required even for special and complicated source code modifications, but instead existing standards such as **XSLT, XPath** and **XQuery** can be used for this purpose.

5. This type of modification permits the setting up of hierarchies, among other things through the possibility for ordered, automated sequential execution (pipelines) of multiple transformations, of patches for example.

6. The transformations can be stored in XSLT files for general reuse, enabling libraries to be produced for example for specific execution sequences.

7. An XML representation of the source code can be stored in an XML database and when necessary easily adapted with the aid

of an XSLT to the individual customer requirements (customization).

8. Through the use of the **XMLSchema** or **DTD** standards or appropriate XSLTs the code can be checked in advance (without compilation) for specific correctness aspects (validated).

9. Standard XML tools can be used for simple processing or visualization and determination of relationships in the code.

10. Permanent XML-based program libraries which support XPath queries can improve the reuse of code through more efficient retrieval of a code or code fragments or templates.

Annex 1:

Listing 1: TestRegistry.java

```
import electric.registry.Registry;
public class TestRegistry
{
  … //further source code in which something has to be changed
}
```

Listing 2: TestRegistry.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <import>
    <dot><dot><name>electric</name><name>registry</name></dot><name>Registry</name></dot>
  </import>
  <class>
    <modifiers><public/></modifiers>
    <name>TestRegistry</name>
    <block>
      …
    <block>
  </class>
</java>
```

Listing 3: VariationPointT1.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- ********************************
       *** copy template    **
   ********************************-->
  <xsl:template match="*">
    <xsl:copy><xsl:apply-templates/></xsl:copy>
  </xsl:template>
  <!-- ********************************************
       ***  Variation Point Transformation 1     **
   ********************************************-->
  <xsl:template match="import">
    <xsl:if test="dot/name='Registry'">
     <import>
       <dot>
         <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
         <name>client</name></dot><name>Call</name>
       </dot>
     </import>
     <import>
       <dot>
         <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
           <name>client</name></dot><name>Service</name>
       </dot>
     </import>
    </xsl:if>
    <xsl:if test="dot/name!='Registry'">
      <xsl:copy-of select="."/>
    </xsl:if>
    …
  </xsl:template>

</xsl:stylesheet>
```

Listing 4: TestRegistry.xjava (*)

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <import>
    <dot>
      <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
      <name>client</name></dot><name>Call</name>
    </dot>
  </import>
  <import>
    <dot>
      <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
        <name>client</name></dot><name>Service</name>
    </dot>
  </import>
  <class>
    <modifiers><public/></modifiers>
    <name>TestRegistry</name>
    <block>
      ...
    <block>
  </class>
</java>
```

Listing 5: TestRegistry.java (*)

```
import org.apache.axis.client.Call; //Axis
import org.apache.axis.client.Service;
public class TestRegistry
{
 ... //further source code in which something has been changed
}
```

Annex 2

Listing 1: TestOutput.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
   <define name="m" value="private">
  <class>
    <modifiers><m/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
      </var>
    </block>
  </class>
</java>
```

Listing 2: TestOutput.xjava*

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><private/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
      </var>
    </block>
  </class>
</java>
```

Listing 3: TestOutput.java

```
private class TestOutput
{
      String m_sWelcome;
}
```

Annex 3

Listing 1: TestCalculator.java

```
public class TestCalculator{
   private int z;

   public void calculate(int x, int y){
        z = x+y;
   }
}
```

Listing 2: TestCalculator.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestCalculator</name>
    <block>
      <var>
         <modifiers><private/></modifiers><type><int/></type><name>z</name>
       </var>
      <method>
        <modifiers><public/></modifiers>
        <type><void/></type>
        <name>calculate</name>
        <params>
          <param><type><int/></type><name>x</name></param>
          <param><type><int/></type><name>y</name></param>
        </params>
        <curly>
          <expr>
            <a>
              <name>z</name>
              <plus><name>x</name><name>y</name></plus>
            </a>
          </expr>
        </curly>
      </method>
    </block>
  </class>
</java>
```

Listing 3: LoggingAspect.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- ********************************
     *** copy template   **
     ********************************-->
<xsl:template match="*">
   <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>

<!-- ********************************
       ***  logging aspect              **
   ********************************-->
<xsl:template match="*[(name()='curly')and(ancestor::method[contains(name,'cal')])]">
  <xsl:copy>
   <expr>
    <paren>
     <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
     <exprs>
      <expr>
```

```
      <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> begin"</xsl:text>
     </expr>
    </exprs>
   </paren>
  </expr>
  <xsl:copy-of select="*" />
  <expr>
   <paren>
    <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
    <exprs>
     <expr>
      <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> end"</xsl:text>
     </expr>
    </exprs>
   </paren>
  </expr>
 </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Listing 4: TestCalculator*.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestCalculator</name>
    <block>
      <var>
         <modifiers><private/></modifiers><type><int/></type><name>z</name>
       </var>
      <method>
        <modifiers><public/></modifiers>
        <type><void/></type>
        <name>calculate</name>
        <params>
          <param><type><int/></type><name>x</name></param>
          <param><type><int/></type><name>y</name></param>
        </params>
        <curly>
          <expr>
            <paren>
              <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
              <exprs><expr>"calculate begin"</expr></exprs>
            </paren>
          </expr>
          <expr>
            <a>
              <name>z</name>
              <plus><name>x</name><name>y</name></plus>
            </a>
          </expr>
          <expr>
            <paren>
              <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
              <exprs><expr>"calculate end"</expr></exprs>
            </paren>
          </expr>
        </curly>
      </method>
    </block>
  </class>
</java>
```

Listing 5: TestCalculator*.java

```
public class TestCalculator{
   private int z;

   public void calculate(int x, int y){
        System.out.println("calculate begin");
```

```
        z = x+y;
        System.out.println("calculate end");
    }
}
```

Annex 4

Listing 1: TestOutput.java

```
public class TestOutput
{
      String m_sWelcome;
}
```

Listing 2: TestOutput.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
      </var>
    </block>
  </class>
</java>
```

Listing 3: State.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<state name="m_sWelcome">
  <value>"hello"</value>
</state>
```

Listing 4: Mixing.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- ********************************
     *** copy template              **
     ********************************-->
<xsl:template match="*">
   <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>
<!-- ********************************
     ***  mixing template           **
     ********************************-->
<xsl:template match="*[(name()='var')and(name='m_sWelcome')]">
 <xsl:copy>
   <xsl:copy-of select="*"/>
     <a>
      <expr><xsl:value-of select="//state[@name='m_sWelcome']/value" /></expr>
     </a>
   </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Listing 5: TestOutput.xjava (*)

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
        <a>
          <expr>"hello"</expr>
        </a>
      </var>
    </block>
  </class>
</java>
```

Listing 6: TestOutput.java (*)

```
public class TestOutput
{
      String m_sWelcome="hello";
}
```

Annex 5

Listing 1A: TestOutput.java

```
public class TestOutput
{
      String m_sWelcome="hello";
}
```

Listing 2A: TestOutput.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
        <a>
          <expr>"hello"</expr>
        </a>
      </var>
    </block>
  </class>
</java>
```

Listing 3A: CodeFragment.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<fragment name="m_sWelcome">
  <expr>
    <paren>
      <dot><name>System</name><name>getProperty</name></dot>
      <exprs><expr>"WELCOME"</expr></exprs>
    </paren>
  </expr>
</fragment>
```

Listing 4A: Patching.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- ********************************
       *** copy template              **
       ********************************-->
  <xsl:template match="*">
    <xsl:copy><xsl:apply-templates/></xsl:copy>
  </xsl:template>
  <!-- ********************************
       ***  patching template         **
       ********************************-->
  <xsl:template match="*[(name()='a')and(ancestor::var/name='m_sWelcome')]">
    <xsl:copy>
      <xsl:copy-of select="//fragment[@name='m_sWelcome']/expr" />
```

```
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Listing 5A: TestOutput.xjava (*)

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
        <a>
          <expr>
            <paren>
              <dot><name>System</name><name>getProperty</name></dot>
              <exprs><expr>"WELCOME"</expr></exprs>
            </paren>
          </expr>
        </a>
      </var>
    </block>
  </class>
</java>
```

Listing 6A: TestOutput.java (*)

```
public class TestOutput
{
      String m_sWelcome=System.getProperty("WELCOME");
}
```